Post Facto Type Extension for Mathematical Programming

Stephen M. Watt

Department of Computer Science
University of Western Ontario
London ON, Canada N6A 5B7
watt@csd.uwo.ca

Abstract

We present the concept of *post facto extensions*, which may be used to enrich types after they have been defined. Adding exported behaviours without altering data representation permits existing types to be augmented without renaming. This allows large libraries to be structured in a clean, layered fashion and allows independently developed software components to be used together. This form of type extension has been found to be particularly useful in mathematical software, where often new abstractions are applicable to existing objects. We describe an implementation of post facto extension, as provided by *Aldor*, and explain how it has been used to structure a large mathematical library.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications—Methodologies; D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages; D.3.3 [*Programming Languages*]: Language Constructs and Features; I.1.3 [*Symbolic and Algebraic Manipulation*]: Languages and Systems—Special-purpose algebraic systems

General Terms Design, Languages

Keywords Aldor, Axiom, Aspect-oriented programming, Symbolic computation, Computer algebra

1. Introduction

As software libraries are extended and combined, it is often desirable to view values of pre-existing types as instances of more general abstractions defined later. This leads either to defining a host of conversions, or to re-writing libraries. Conversions may be either implicit or explicit, but in either case the programmer must be aware of them and use compilers that can optimize them. Re-writing libraries to endow pre-existing types with later-defined semantics is time-consuming and decreases modularity.

This paper presents a programming language solution to this problem. The solution, "*post facto extension*," is a specialized instance of what is today known as aspect-oriented programming, and it has proven highly effective in structuring mathematical libraries for *Aldor* [1, 2, 3, 4]. Language support for post facto extension can be readily added to object oriented or abstract datatype programming languages without the complexity of full support for general aspect-oriented programming.

This work has been motivated by the design of software for computer algebra, an area concerned with answering mathematical problems in terms of symbolic expressions rather than numbers. In mathematics, as in software development, one of the principal activities is that of generalization. By expressing problems more abstractly, it is possible to make greater re-use of previous work. From this point of view, it is quite natural to view previously defined quantities as special instances of newly defined abstractions.

A simple example illustrates this point: Suppose we are developing a library, and one of the types is `Integer`. We provide this type with the basic arithmetic operations, $+$, $-$, $\times$, $=$, $<$, *etc*. Later, `gcd` and `lcm` are added to the library. Should the type `Integer` be modified to export these operations, or should they remain as independent functions? If additional arithmetic types are added, it may be desirable to add a `Ring` abstraction, from which all types with suitable arithmetic can inherit. Types that could provide the `Ring` interface include square matrices, polynomials, quotient fields, complex numbers and the integers. Should `Integer` be modified to export `Ring`? If `Integer` *is not* modified, then `Integer` values cannot be used where elements of a `Ring` are required. It is then necessary to introduce a new type and provide conversions. As more abstractions are added, many conversions are used either explicitly or implicitly and code becomes cumbersome and inefficient. If `Integer` *is* modified, then a new dependency is created and previously complete components must be re-tested. It is also possible that adding this behaviour to the type will break third-party uses of the library. When more than one library is involved, this problem is exacerbated.

The problem of dealing with new abstractions for existing objects is by no means restricted to mathematical computation. It arises whenever multiple libraries provide different functionality for basic types in an object-oriented environment. An example would be when different libraries provide string manipulation, regular expression matching and higher-level text operations. Situations such as this are now well-understood in the aspect-oriented programming community. We have found this problem to be particularly acute in the construction of computer algebra software: *In this setting, it is the usual case that most basic types are instances of many later-defined abstractions.* Moreover, since there is wide agreement on numerous mathematical abstractions, it is quite natural to expect the objects of one library to simply work in other libraries.

The situation where we first noticed the problem was in the construction of libraries for the *Axiom* [5] computer algebra system. Given that the basic arithmetic types needed to participate in advanced mathematical operations, there seemed to be no way to de-

DSAL 2006 Portland OR, USA.

fine a fixed core language with a few basic types and an evolving library of advanced functionality. During this period, the present author was responsible for the design and implementation of the programming language to be used for libraries to extend the *Axiom* system. We were thus in the fortunate position to consider programming language solutions to problems that arose in library design.

Our solution to the problem of dealing with new abstractions for existing components relies on a key observation: *Although it is desirable to add new interfaces to mathematical types after they have been defined, it is usually not desirable to change the representation of values.* We found it was almost always the case that any new operations required by the new interfaces could be defined in terms of existing exported behaviour without any change to the object representation. This led to the idea that existing values could participate in new interfaces without any changes to the objects at all. Instead, higher-order operations on the types could add the desired behaviours. This is the basic idea of what we call "post facto extension" of types.

We have explored this idea in our design of the *Aldor* programming language and have found it to be quite effective in cleanly structuring complex mathematical libraries with many rich relations among the types. In *Aldor*, the expression of post facto extensions is quite simple. From the programmer's point of view, there is little required to use them effectively. We believe that these ideas may prove useful in areas outside of mathematical programming and therefore should be more widely known.

This paper presents the main ideas of post facto type extension and describes how it has been used to structure complex libraries: Section 2 outlines the main ideas of *Aldor* and its type system. Section 3 describes structural problems that were observed in building mathematical libraries for *Aldor*. Section 4 then presents our solution, *post facto extensions*. Section 5 explains some of the ways that post facto extensions can be used in structuring large libraries. We present our conclusions in Section 6.

2. *Aldor* and Its Type System

Aldor [1, 2, 3, 4] is a programming language originally intended to provide compiled libraries for computer algebra. The design of the language tries to balance high-level mathematical expressivity with the possibility of compilation to efficient machine code so large symbolic and numeric problems can be treated. There are several aspects to the *Aldor* language that are intended to provide support for mathematical programming, but which are somewhat unusual. We outline these below.

Types and functions are first-class values. This means that they may be created and used dynamically, providing representations for mathematical sets and functions.

The type system has two levels. Each value belongs to some unique type, known as its *domain*, and the domains of values can be delcared statically. Domains themselves belong to the domain `Type`. Domains may additionally belong to type *categories* that specify additional properties. In particular, categories may specify that a domain must export certain operations or that some operations have default implementations. Categories fill the role of interfaces or abstract base classes of other languages, and may be viewed as sub-types of the domain `Type`. Category membership can be asserted at compile time and tested at run time.

The language is not object-oriented. There are a number of aspects of object-oriented programming that make it awkward to use in an algebraic setting:

The first problem is that object-oriented languages favour a programming style where objects maintain state and the execution of a program consists of calling methods to change that state. Mathematical programming is more suited to a functional style, where one works with values and functions compute new values based on their arguments and where values are seldom, if ever, modified.

The second problem is that, in an object-oriented world, binary operations do not inherit in a natural way. In mathematics it is quite common to have functions that are homogeneous on their arguments, for example $+$, $\times$, $-$, $=$ and $<$. To illustrate the difficulty with object-oriented inheritance, suppose we have a base class B with a method plus, used as `a.plus(b)` to add a value of type B to an object of type B and yielding new value of type B. That is, `plus` : $B \times B \to B$. If class D is derived from class B then it will have `plus` : $D \times B \to B$. This problem was already noted by Barbara Liskov as arising in the design of *CLU* [6] and is cited as one of the reasons that the language was based on abstract data types rather than objects.

The third problem is related to the second. Class-based inheritance does not provide sufficient static type checking for multiple-argument functions. To illustrate, suppose that a base class B provides the abstraction of multiplication $\times : B \times B \to B$ and that classes D_1 and D_2 are derived from it. We wish to ensure statically that the multiplications $D_1 \times D_1$ and $D_2 \times D_2$ are allowed, but that the multiplications $D_1 \times D_2$ and $D_2 \times D_1$ are disallowed. In an object-oriented world, however, the inhomogeneous multiplications would be allowed by virtue of the inherited multiplication defined in class B.

An example can illustrate this last point. In *Aldor* one can define a category `Semigroup` to capture the abstraction of a homogeneous multiplication, and the domains `DoubleFloat` and `Permutation` could be declared to belong to this category.

```
Semigroup:   Category == with { *: (%, %) -> % }
DoubleFloat: Join(Semigroup, ...) == ...
Permutation: Join(Semigroup, ...) == ...
```

This causes the declared domains to export a suitable multiplication. For example, `DoubleFloat` will have an exported operation "`*`" that takes two `DoubleFloat` values and returns a third. If `x` and `y` are declared to be of type `DoubleFloat` and `p` and `q` are declared to be of type `Permutation`, then it will be possible to multiply `x*y` and `p*q`, but not `x*p`. This difference may be summarized by the following relations:

$$\left.\begin{array}{lllll} \texttt{x,y} & \in & \texttt{DoubleFloat} & \subset & \texttt{Semigroup} \\ \texttt{p,q} & \in & \texttt{Permutation} & \subset & \texttt{Semigroup} \end{array}\right\}\text{OOP}$$

$$\left.\begin{array}{lllll} \texttt{x,y} & \in & \texttt{DoubleFloat} & \in & \texttt{Semigroup} \\ \texttt{p,q} & \in & \texttt{Permutation} & \in & \texttt{Semigroup} \end{array}\right\}\text{Aldor}$$

In an object orient world `x` and `p` belong to a common inherited class, but in *Aldor* they do not.

Dependent types are fully supported. *Aldor* obtains the capabilities of object-oriented programming through the use of dependent types. Tuples may have components whose value determines the type of other components and mappings may have return types that depend on the values of parameters. As an example, consider the following declaration:

```
f: (n: Integer, m: SquareMatrix(n, Integer))
   -> List IntegerMod(n)
```

Here we suppose that `SquareMatrix(n, Integer)` is the type of $n \times n$ square matrices with integer entries and that `IntegerMod(n)` is a type representing the integers modulo n. The *types* of the second argument and of the return value of `f` depend on the *value* of the first argument. If the first argument is 3, then the second argument must be a 3×3 matrix and the result will be a list of integers modulo 3.

Dependent types are particularly useful when some of the components are themselves types. For example, we may define

```
prodl: List Record(S: Semigroup, s: S) == [
   [DoubleFloat, x],
   [Permutation, p],
   [DoubleFloat, y]
]
```

Here each element of the list consists of a type and a value belonging to that type. By specifying that the type belong to a particular category, we are able to determine statically what operations are supported on the values. In *Aldor*, use of dependent types and inheritance in the category hierarchy take the place of objects and inheritance in the class hierarchy.

Parametric polymorphism is provided by category- and domain-producing functions. With types as first class values and dependent types fully supported, functions producing types take the place of templates in other languages. For example we may write

```
define Module(R: Ring): Category == Ring with {
    *: (R, %) -> %
}

Complex(R: Ring): Module(R) with {
    complex: (%, %) -> R;
    real: % -> R;
    imag: % -> R;
    conjugate: % -> %;
    ...
} == add {
    Rep == Record(real: R, imag: R);
    ...
}
```

Here, `Module` is a function that take a type parameter, `R`, belonging to the category `Ring` and returns a category as its result. The form `Ring with {*: (R, %) -> R}` constructs the category to be returned as being the category `Ring` extended with one additional operation. The symbol `%` in the category expression refers to the domain that exports the category. If `D: Module(T)`, then `D` exports `*: (T, D) -> D`. The keyword `define` affects the publicly visible information about `Module` that will be visible about compilation units. It allows not only its type, `(R: Ring) -> Category`, but also its value, `(R:Ring) +-> Ring with {*:(R,%) -> %}`, to be publicly visible.

The second definition declares `Complex` to have a dependent mapping type, `(R: Ring) -> Module(R) with...`. That is, `Complex` takes a type-valued parameter `R` that belongs to the category `Ring` and returns a type-valued result that belongs to the category `Module(R) with...`. The body of the function definition (the part after "==") is a form that constructs a domain.

Category- and domain-producing expressions may be conditional. *Aldor* provides conditional inheritance, allowing types to be formed differently according to run-time conditions. For example, we may write

```
UnivariatePolynomial(R: Ring): Module(R) with {
    coeff: (%, Integer) -> R;
    monomial: (R, Integer) -> %;

    if R has Field then EuclideanDomain;
    ...
} == add {
    ...
}
```

That is, if the type parameter `R` to `UnivariatePolynomial` not only belongs to the category `Ring` but also belongs to the category `Field`, then the type `UnivariatePolynomial(R)` also belongs to the category `EuclideanDomain`.

Post facto extensions. *Aldor* allows domains to be extended to belong to new categories after they have been initially defined. These allow domains to be defined in a layered fashion, separating issues and eliminating dependencies, while providing rich function. These are the focus of the present paper and are described in more detail in Section 4.

Aldor grew out of an earlier language by Jenks and Trager [7] that already used the idea of domains and categories. This language was the original library language for the *Axiom* system (then known as *Scratchpad II*), and inspired a number of other projects for computer algebra languages, including *Newspeak* [8] and *Views* [9].

3. Problems in Library Design

We now describe a certain problems that we observed in building the first *Aldor* libraries. We describe some of these problems using the terminology of *Aldor*, but their translation into other programming languages should be straightforward. Later we show how these problems are solved with post facto extensions.

Old domains and new categories. We can now revisit the example of the introduction using more precise language: In building libraries for mathematical computation, it is quite normal to define new categories and to find that existing domains could be made to belong to them. Many of the most basic types, such as `Integer`, `IntegerMod(p)`, `Fraction(R)`, `Complex(R)`, `Matrix(n,m,R)` and `UnivariatePolynomial(R)`, have a wealth of mathematical properties and are often potential instances of newly defined categories. The same thing is true for floating point types if one is willing to overlook the fact that they are not exactly associative.

In building the basic *Aldor* libraries, there was the choice of whether to modify these basic domains to belong to all the applicable categories defined in the standard libraries, or whether to maintain modularity. On the one hand, even within the standard libraries, modularity was desirable. Certain types, such as `Integer` and `Boolean` must be known in the language definition and it would be injudicious to therefore have to fix an intricate hierarchy of algebraic categories as part of the basic language. Even if these basic domains were modified to belong to all the applicable categories in the standard libraries, then the problem of membership in categories from new libraries would still exist. On the other hand, if the basic domains were not made to belong to the categories of the standard libraries, then values belonging to these domains could not be used by any of the advanced functions. The solution of having a basic and an elaborated version of each type would lead to code filled with distracting explicit conversions or subtly dangerous implicit conversions.

Difficulties with multiple libraries. Commonly, application must work with objects that inherit from base classes or interfaces from independent libraries. There is the problem, however, that objects returned by methods of one library are not suitable for use in calls to methods of other libraries. One solution is for the application to build its objects as compound structures containing component objects from the separate libraries. In this case conversions and constructors are used to move between the types. Sometimes an application will define a new base class for its hierarchy that inherits from both libraries as a way to deal with this situation. Then clients of the application that require yet other third libraries must repeat the process. This is really just another instance of old types lacking new interfaces. except in this case the interfaces come from separate libraries and there is no real possibility of integrating the set of types.

Large dependency sets in libraries. In many programming languages dependencies can arise among components because types refer to each other in their definitions. In *Aldor* and certain other languages, dependencies can also arise because types refer to each other in their *type*. We give a simple example. Suppose we have the following declarations:

```
define AbelianGroup: Category == with {
    +: (%, %) -> %;
    *: (Integer, %) -> %;
    ...
}

define DifferentialRing: Ring with {
    diff: % -> %;
}

Integer: Join(DifferentialRing, ...) == ...
```

That is, the domain `Integer` is declared to (trivially) belong to the category `DifferentialRing` so that it be possible to construct differential operators and other structures with integer coefficients. The problem is that the type `Integer` appears in the definition of `AbelianGroup`. Because of this, all domains that belong to `AbelianGroup` have an indirect and undesired dependency on `DifferentialRing`.

In compiling programs we may wish to verify that expressions have well-defined type, to verify that types are well-formed and to perform type inference. The dependencies that arise through the types of types can lead to large systems requiring fixed-point analysis. This not only imposes technical constraints on the type system, it also requires careful compiler design to avoid long compilation times for simple programs.

This form of dependency has been seen to be a practical problem. In the design of the Axiom system, basic mathematical types were endowed with all appropriate advanced algebraic interfaces. This led to an almost complete inter-dependency among the interface specifications. Doing a complete type checking of the library interfaces took several days, and this led to a reluctance to modify the library.

Complex conditionalization. While conditional category membership is one of the more useful features of the *Aldor* language and its predecessors, it is also subject to difficulties when new categories are used. We illustrate this with the domain-constructing function `DirectProduct(n, S)` which constructs the type of n-tuples of values from the type `S`.

```
DirectProduct(n: Integer, S: Set): Set with {
    component: (Integer, %) -> S;
    new:       Tuple S -> %;

    if S has Semigroup then Semigroup;
    if S has Monoid then Monoid;
    if S has Group then Group;
    ...
    if S has Ring  then Join(Ring, Module(S));
    if S has Field then Join(Ring, VectorField(S));
    ...
    if S has DifferentialRing then DifferentialRing;
    if S has Ordered then Ordered;
    ...
} == add {
    ...
}
```

Here we see that the set of categories satisfied by `DirectProduct(n, S)` depends very much on the categorical properties of the argument `S`. The direct product inherits from many, but not all, of the categories satisfied by `S`. For example, if `S` is a `Monoid`, then so is `DirectProduct(n,S)`. The same is true for many other categories. Sometimes `DirectProduct(n,S)` does *not* belong to the categories satisfied by `S`. For example, if `S` is a `Field` then `DirectProduct(n,S)` is not. Sometimes the opposite is true: sometimes `DirectProduct(n,S)` belongs to *additional* categories by virtue of the categorical properties its argument. This occurs, for example, when `S` is a `Ring`.

This example serves to make two points: First, we see that the categorical properties of a domain-constructing function can be quite complex and depend very much on the specific nature of the type constructor — it is not possible to describe this behaviour with a few simple universal rules. Second, we see that certain constructors are open-ended in their conditionalization requirements — whenever new categories are added to the environment, it is likely the constructor should be augmented.

4. Post Facto Extensions

Our solution to the problems we have outlined is to provide a mechanism for domain-valued expressions to have their meaning augmented with additional categories. This is achieved by allowing names bound to domains and domain-producing functions to have additional definitions and by providing rules by which the multiple meanings visible in a given scope are to be combined. We explicitly note that the representation of values belonging to the augmented domains does not change. All that is different is that the domain to which they belong is made to belong to additional categories, and consequently support more operations.

Extending domains. If D is a domain-valued constant, then its meaning may be extended with a definition of the form

`extend` D`:` C `==` E

This declares the D to belong to the category given by C in addition to whatever other categories it belongs in the current scope. In general, belonging to this new category may require D to provide new exports. The expression E gives the implementation of these new exports in terms of previously exported operations. The keyword `extend` is required so that the definition is not taken to be an independent, overloaded meaning.

To illustrate, the domain `Integer` may be made to belong to the category `DifferentialRing` by providing the following extension

```
extend Integer: DifferentialRing == add {
    diff(n: Integer): Integer == 0;
}
```

Separately, `Integer` may be made to belong to the category `ConvertibleTo(MathML)` by providing the extension

```
extend: Integer: ConvertibleTo(MathML) == add {
    convert(n: Integer): MathML == mi(n)
}
```

Named domains can in this way have different behaviours added as needed. When an existing domain is used with a new library, then a set of extensions can be provided to make the domain belong to whichever categories are desired. The programmer is free to organize the extensions in any suitable manner. In a scope where a domain-valued constant is used, its type is taken to be the `Join` of all the categories of the visible extension definitions and its value is taken to be the `add` of all the expressions from the extension definitions. That is, if the visible definitions are

```
N: C0 == A0;
extend N: C1 == A1;
...
extend N: Cn == An;
```

then the domain used will be formed as

```
N: Join(C0,C1,...,Cn) ==
A0 add A1 add ... An add {}
```

Extending functions. Domain-producing functions may be extended by providing an additional function definitions, marked with `extend`. An extension of a domain-producing function must have arguments with the same domains as the corresponding arguments of the original function. Normally, however, one or more of the arguments will have different subtype properties. (This includes the case where domain-valued arguments are declared to belong to different categories.) The declared return type of the extension function is taken to be an additional category to which the resulting domain will belong. To illustrate, we rewrite the `DirectProduct` example using extensions as shown in Figure 1. It would not normally be the case that the extensions would be given together as shown here. More often the extensions would either be placed together with the category definitions or be grouped in some way (*e.g.* extensions necessary to make domains of library 1 work with library 2).

In a scope where a domain-producing function constant is used, the original function value and all of the visible extensions are combined to produce the function that is actually used. This allows proper behaviour of function-valued names. So, for example, extended domain-producing functions may be passed as parameters, saved as values *etc*, and later used as desired.

If the visible function definition and extensions for F are

```
F(a1: T01,...,ak: T0k): R0 == A0
extend F(a1: T11,...,ak: T1k): R1 == A1
...
extend F(a1: Tn1,...,ak: Tnk): Rn == An
```

this is equivalent to the definition

```
F(a1:Meet(T01...Tn1),...,an:Meet(T0k...Tnk)): with {
    if a1 ∈ T01 and ... and ak ∈ T0k then R0;
    if a1 ∈ T11 and ... and ak ∈ T1k then R1;
    ...
    if a1 ∈ Tn1 and ... and ak ∈ Tnk then Rn;
} == add {
    if a1 ∈ T01 and ... and ak ∈ T0k then A0;
    if a1 ∈ T11 and ... and ak ∈ T1k then A1;
    ...
    if a1 ∈ Tn1 and ... and ak ∈ Tnk then An;
}
```

Here the symbol "$\in$" is interpreted to be a subtype test for the corresponding base domain. In particular, when $\mathtt{T}_{ij}$ is a category "$\in$" means "`has`." If $\mathtt{T}_{0i} = \mathtt{T}_{1i} = \cdots = \mathtt{T}_{ni}$ then the i-th test can be omitted.

These rules are applied recursively, with suitable interpretation of `Meet`, `Join` and `add`, so that curried domain-producing functions are handled naturally.

Implementation. In *Aldor* data values are not necessarily self-identifying, but each expression has a unique well-defined domain. Operations on data values are in principle extracted during execution from these domain objects and it is the compiler's responsibility to ensure that all the necessary domain objects are available at known locations at run-time. Post-facto extension of domains is implemented by constructing composite domain objects. Post-facto extension of functions is implemented by combining functions as described above. One of the most important aspects of the implementation of post facto extension is the static optimization of extension compositions, determining which functions should be called during execution. This enables a number of further optimizations, resulting in relatively efficient code.

Post facto extension can also be implemented in an object-oriented environment. In this case it is necessary to modify data structures representing class objects. These classes (including virtual function tables) are usually accessed through the member objects so there is the added complexity of matching the lifetime of the post facto extensions with their scope.

Relation to other work. Our design of post facto extensions makes use of the idea of mixins, from the Flavors system [10], applied to type-producing functions. This allows a separation of concerns in the creation and use of first class type objects, as described in [2]. The result gives a specialized form of what has come to be known as aspect-oriented programming [11], applicable to parameterized and non-parameterized types. If we view contant domains as nullary domain-producing fuctions, we may view post-facto extension as providing scoped point cuts associated to domain constructors. In the non-parametric case, a similar effect can be achieved with open classes [12]. The use of type categories in *Aldor* allows the compiler to perform various optimizations, as described in [1], to eliminate function look-up and perform in-lining where possible, taking into account post facto extensions.

5. Use in Library Design

We now have a dozen years' experience in the use of post facto extensions for structuring mathematical libraries for *Aldor*. This section describes some of the ways in which we have found it to be useful.

Uniform treatment of raw types and object types. Many programming languages make a distinction between "raw types" and "object types." This distinction does not exist in *Aldor*. The *language* defines a set of standard types and the *library* endows them with operations. All basic domains are initially defined simply as data representations. All primitives are given as independent operations, provided by the `Machine` package. For example, we have

```
Boolean:       Type == add {};
Integer:       Type == add {};
DoubleFloat:   Type == add {};
...
```

and the `Machine` package provides primitives for arithmetic on values of these types. Later, these types are extended by the standard library to have a richer structure.

Layering large libraries. We have found it useful to be able to build large libraries in layers, with fewer dependency cycles. In bootstrapping the Standard *Aldor* Library, we have the following layers:

1. *Basic types without operations.* The basic types are simply declared to be types, and the data representation is implicit in the use of available machine primitives on these types.
2. *Basic types with representation.* The basic types are extended to themselves export the relevant primitives. They may now be treated as types with an opaque representation. As the basic types are extended with operations, they may refer to each other in the signatures of their exports. For example, we may have

```
extend Boolean: with {
    =: (%, %) -> Booelean;
    convert: % -> String;
    ...
} == ...

extend Integer: with {
    =: (%, %) -> Boolean;
    <: (%, %) -> Boolean;
    convert: % -> String;
    ...
} == ...
```

```
DirectProduct(n: Integer, S: Set): Set with {
    component: (Integer, %) -> S;
    new:       Tuple S -> %;
} == add { ... }

extend DirectProduct(n: Integer, S: Semigroup): Semigroup == ...
extend DirectProduct(n: Integer, S: Monoid): Monoid == ...
extend DirectProduct(n: Integer, S: Group): Group == ...
...
extend DirectProduct(n: Integer, S: Ring): Join(Ring, Module(S)) == ...
extend DirectProduct(n: Integer, S: Field): Join(Ring, VectorField(S)) == ...
...
extend DirectProduct(n: Integer, S: Field): Join(Ring, VectorField(S)) == ...
extend DirectProduct(n: Integer, S: DifferentialRing): DifferentialRing == ...
extend DirectProduct(n: Integer, S: Ordered): Ordered == ...
...
```

Figure 1. `DirectProduct` defined using extensions

```
extend String: with {
    =: (%, %) -> Boolean;
    #: % -> Integer -- Length.
    ...
} == ...
```

These may be compiled without having to resort to a multi-type fixed-point determination in type inference.

3. *Definition of constructed types.* The library uses the primitive types to construct a richer set of useful types, such as linked lists, hash tables, I/O abstractions, *etc*.
4. *Types with useful categories.* The Standard Library defines a number of categories, and the basic and constructed types are extended to belong to them as appropriate.

The Algebra Library is built on top of the Standard Library as follows:

5. *Mathematical categories.* The Algebra Library defines a rich categorical structure with categories corresponding to many of the standard algebraic abstractions. These include abstractions for the concept of group, ring, euclidean domain, field, module, algebra, *etc.*
6. *Extension of the basic types.* The arithmetic types of the Standard Library are extended to belong to all the appropriate categories from the Algebra Library.
7. *Definition of mathematical domains.* The library defines a set of common mathematical domains, such as polynomials, matrices, quotients, finite fields and so on.

A number of more sophisticated mathematical libraries are built on top of the Algebra Library, and these extend the types of the Standard Library and Algebra Library, as appropriate.

This layering allows the elimination of cyclic dependencies in the design of the libraries and allows the libraries to be built and tested in a modular fashion. It does this without compromising the rich set of behaviours desired for the basic types.

Combined use of multiple libraries. With post facto extensions it is quite easy to use multiple, independently developed libraries without a host of data conversions. The application programmer decides which categories from the various libraries will be important and extends the necessary types to export them. Having done this, the values computed by one library may be readily used in the other libraries without conversion.

Separation of concerns. With post facto extensions it is straightforward to separately implement various independent aspects of domains. This is one of the standard goals of aspect-oriented programming. For example, one set of extensions can provide algebraic algorithms, while another set of extensions provide translations to TEX and a third set of extensions provide translations to MathML. The code for each set of extensions can be separately developed, tested and maintained.

Adding callback algorithms to parameters. One of the difficulties with generic programming is that there are often specialized algorithms that apply over certain domains. In C++ this is handled by template specialization and is resolved statically. However, in *Aldor* types may be constructed dynamically so we need some other mechanism to access specialized algorithms. Post facto extension, combined with conditional category tests, allows generic code to use special purpose algorithms, when applicable, without revising library components.

We illustrate this point with an example from linear algebra. Such a package can be defined generically over any commutative ring. More efficient algorithms may be used, however, when the ring is known to be an integral domain or a field. We may thus assemble these algorithms into a package as follows:

```
LinearAlgebra(R:CommutativeRing, M:MatrixCategory R):
with {...} == add {
    local Elim: LinearEliminationCategory(R, M) == {
        R has Field =>
            OrdinaryGaussElimination(R, M);
        R has IntegralDomain =>
            TwoStepFractionFreeGaussElimination(R,M);
        DivisionFreeGaussElimination(R, M);
    }

    determinant(m:M):R == determinant(m)$Elim;
}
```

Certain coefficient rings may support efficient specialized algorithms. For example, we may want to compute over the integers using Chinese remaindering. However, we do not want to have to modify the `LinearAlgebra` package whenever a new method is incorporated into the library. We therefore define a category that a ring can implement to provide linear algebra algorithms over itself:

```
LinearAlgebraRing: Category == with {
    determinant: (M:MatrixCategory %) -> M -> %;
    rank:        (M:MatrixCategory %) -> M -> Integer;
    ...
}
```

We make one modification to the `LinearAlgebra` package to take advantage of special-case algorithms carried in a `LinearAlgebraRing` view: we replace the `determinant` function with

```
determinant(m:M):R == {
    if R has LinearAlgebraRing then
        determinant(M)(a)$R;
    else
        determinant(m)$Elim;
}
```

When we have special algorithms for some domain, we extend the domain to know about them:

```
extend Integer: LinearAlgebraRing == add {
    determinant(M: MatrixCategory %)(m: M): % ==
        ChineseRemainderingDeterminant(M, m);
    rank(M: MatrixCategory %)(m: M): % ==
        ChineseRemainderingRank(M, m);
    ...
}
```

Whenever we use the `LinearAlgebra` package, it will use the designated algorithm even if the coefficient ring is determined dynamically.

The technique of using post facto extensions to endow domains with special- case algorithms has been used in the in the construction of the $\sum^{IT}$ library [13]. The notion of rings knowing how to perform operations in structures over themselves has been explored earlier in relation to composing factorization algorithms [14].

6. Conclusions

We have examined a number of problems that arise in the construction of software libraries. These are all related to question of whether existing types should be updated with new abstractions as libraries grow or are used together. We have noted that the problem is particularly accute in computer algebra, where it is quite usual that pre-existing types can satisfy newly defined abstractions. We have observed, however, that in mathematical programming new abstractions do not usually require any change in data representation in order to be applied. Based on this observation, we have proposed the notion of *post facto extension* of types and of type producing functions. This provides a specialized instance of aspect-oriented programming that has proven particularly effective for mathematical computing.

Acknowledgments

The author would like to thank Samuel S. Dooley for his assistance with the first implementation of post facto extensions in the $A^\sharp$ compiler at IBM Research. He would also like to express his gratitude to the late Manuel Bronstein for the `LinearAlgebraRing` example.

References

[1] Watt, S.M., Broadbery, P.A., Dooley, S.S., Iglio, P., Morrison, S.C., Steinbach, J.M., Sutor, R.S.: *A first report on the $A^\#$ compiler*, Proc. 1994 International Symposium on Symbolic and Algebraic Computation, pp. 25–31, ACM Press.

[2] Watt, S.M., Broadbery, P.A., Dooley, S.S., Iglio, P., Morrison, S.C., Steinbach, J.M., Sutor, R.S.: *AXIOM Library Compiler User Guide*, The Numerical Algorithms Group Ltd, Oxford 1994.

[3] Watt, S.M., *Aldor*, in *Handbook of Computer Algebra*, Grabmeier, Kaltofen and Weispfenning (editors), Springer Verlag 2003, pp. 265–270.

[4] *Aldor User Guide*, `http://www.aldor.org` (2003).

[5] Jenks, R.D., Sutor, R.S.: *Axiom: The Scientific Computation System*, Springer Verlag (1992).

[6] Liskov, B. A history of CLU, Proc. Second ACM SIGPLAN conference on History of programming languages, pp. 133–147, ACM Press (1993).

[7] Jenks, R., Trager, B.: *A language for computational algebra*, Proc. 1981 ACM Symposium on Symbolic and Algebraic Computation, pp. 6–13, ACM Press.

[8] Foderaro, J.K.: *The Design of a Language for Algebraic Computation*, Ph.D. Thesis, UC Berkeley, 1983.

[9] Abdali, S.K., Cherry, G.W., Soiffer, N.: *A Smalltalk system for algebraic manipulation*, Proc. 1986 Object Oriented Programming Systems Languages and Applications, pp. 277–283, ACM Press.

[10] Moon, D.A.: *Object-oriented programming with flavors*, Proc. 1986 Object Oriented Programming Systems Languages and Applications, pp. 1–8, ACM Press.

[11] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., Irwin, J.: *Aspect-Oriented Programming*, Proc. European Conference on Object-Oriented Programming, LNCS 1241 pp. 220-242, Springer Verlag (1997).

[12] Millstein, T., Chambers, C.: *Modular Statically Typed Multimethods*, Proc. European Conference on Object-Oriented Programming, LNCS 1628 pp. 279-303, Springer Verlag (1999).

[13] Bronstein, M.: *$\sum^{IT}$: A strongly-typed embeddable computer algebra library*, Proc. DISCO'96, LNCS 1128 pp. 22-33, Springer Verlag.

[14] Davenport, J., Gianni, P., Trager, B.: *Scratchpad's view of algebra II: a categorical view of factorization*, Proc. 1991 International Symposium on Symbolic and Algebraic Computation, pp. 32–38, ACM Press.